SC09-1129-01

C Compiler User's Guide For MVS, MVS/XA

SC09-1129-01

C Compiler User's Guide For MVS, MVS/XA

Second Edition (October 1987)

This edition applies to the IBM C for System/370 Program Offering (Products 5713-AAG and 5713-AAH).

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to

IBM Canada Ltd.
Information Development,
Department 849,
1150 Eglinton Ave. East
North York, Ontario, Canada. M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright Whitesmiths, Ltd. 1978, 1987
IBM C Compiler by Whitesmiths, Ltd.

IBM is a registered trademark of International Business Machines Corporation, Armonk, NY.

MVS, MVS/XA

Contents

Chapter 1: C Compiler Overview

Introduction ..... 1 - 1
Compiler Architecture ..... 1 - 1
Linking ..... 1 - 1
Listings ..... 1 - 2

Chapter 2: Entering C Programs

Rules for C Program Source Files ..... 2 - 1
Entering Programs with the ISPF Editor ..... 2 - 1
Entering Programs under TSO ..... 2 - 2

Chapter 3: Compiling Programs

Compiler Options ..... 3 - 2
Invoking the Compiler under TSO ..... 3 - 5
Invoking the Compiler in a Batch Job ..... 3 - 6
Invoking the Compiler under ISPF ..... 3 - 7

Chapter 4: Linking and Loading Programs

Using the Linkage Editor under ISPF ..... 4 - 1
Using the Linkage Editor and Loader under TSO ..... 4 - 1
Using the Linkage Editor and Loader in a Batch Job ..... 4 - 2
Linking for Execution Under IMS ..... 4 - 4
Controlling Stack Size ..... 4 - 4
Reentrant Programs ..... 4 - 5
Creating Reentrant Programs ..... 4 - 5
How the Compiler Processes a Reentrant Program ..... 4 - 5
How RLINK Processes a Reentrant Program ..... 4 - 6
Datasets Required by RLINK ..... 4 - 6
Using the Link Pack Area Under MVS and MVS/XA ..... 4 - 8
Restrictions on Making Programs Reentrant ..... 4 - 8

Chapter 5: Executing Programs

Executing Programs Under ISPF ..... 5 - 1
Executing Programs Under TSO ..... 5 - 1
Executing Programs in a Batch Job ..... 5 - 2

Chapter 6: Debugging Programs

Run the Program to Termination ..... 6 - 3
Stepping One Source Line at a Time ..... 6 - 3
Stepping Through Multiple Source Lines ..... 6 - 3
Exit Debugging ..... 6 - 4
Examining Data Objects ..... 6 - 4
Displaying Current Source Line and File Name ..... 6 - 5
Executing Multiple Debugger Commands ..... 6 - 5
Logging Debugger Output ..... 6 - 6
Locating Source Code in Other Datasets ..... 6 - 6

Contents

Setting Breakpoints ..... 6 - 6
Deleting Breakpoints ..... 6 - 7
More Debugger Commands ..... 6 - 7
Moving in Stack Frames ..... 6 - 7
Other Examine/Modify Commands ..... 6 - 8
Formatting Output from the Debugger ..... 6 - 8
Printing the Contents of Data Objects ..... 6 - 9
Type Casts ..... 6 - 9
Summary of Location Specifiers ..... 6 - 10

Chapter 7: Maintaining Libraries
Creating Libraries ..... 7 - 1
Adding a Member to a Library ..... 7 - 1
Deleting a Member of a Library ..... 7 - 2
Replacing a Member of a Library ..... 7 - 3
Listing Members of a Library ..... 7 - 3

Chapter 8: Interfaces with Other Languages
Calling Assembly Language Functions From C ..... 8 - 1
Mapping C Identifiers to External Names ..... 8 - 1
Transferring Control to Another Function ..... 8 - 2
Preserving Registers ..... 8 - 2
Passing Argument Data Objects ..... 8 - 3
Using the Stack ..... 8 - 4
Returning a Value ..... 8 - 5
Returning Control to the Calling Function ..... 8 - 5
Calling Functions Written in Other Languages from C ..... 8 - 5

Appendix A: Compiler Command Summary ..... A - 1

Appendix B: Debugger Command Summary ..... B - 1
Format Specifiers ..... B - 2
Type Casts ..... B - 2

Appendix C: EBCDIC Character Table ..... C - 1

Index ..... X - 1

Preface

The *C Compiler User's Guide for MVS, MVS/XA* (SC09-1129) is a reference guide for programmers writing C programs under MVS and MVS/XA on the System/370 architecture. It provides an overview of how the compiler works, and explains how to compile, link, execute, and debug programs using the System/370 C Compiler under MVS. It assumes that you are familiar with MVS and MVS/XA.

You can find information about the C language and the C runtime library in your *C Language Manual* (SC09-1128).

Organization of this Manual

This manual is divided into eight chapters:

Chapter 1, "C Compiler Overview," describes the individual programs that make up the compiler and their interrelationship.

Chapter 2, "Entering C Programs," explains the process of creating a C source file under ISPF and covers important compiler restrictions about line lengths and file characteristics.

Chapter 3, "Compiling Programs," describes the compilation process under TSO and ISPF, or from a batch job, and how to take advantage of the many features and options available.

Chapter 4, "Linking and Loading Programs," explains how to use your linkage editor and loader to run programs from ISPF and TSO, or with a batch job.

Chapter 5, "Executing Programs," discusses how to run your program from ISPF or TSO, or with a batch job. Procedures for passing data to a program are also described.

Chapter 6, "Debugging Programs," describes debugging techniques and the use of the C source level debugger included with the System/370 C compiler.

Chapter 7, "Maintaining Libraries," explains how to create libraries, and add or delete members of a library.

Chapter 8, "Interfaces with Other Languages," documents how to call a C function from within an assembly language routine, and how to write an assembly language routine that can be called as a C function.

Appendix A, "Compiler Command Summary," lists all the compiler options and what they do.

Appendix B, "Debugger Command Summary," lists all the debugger commands and command formats and what they do.

Appendix C, "EBCDIC Character Table," is a table of all EBCDIC characters and their hexadecimal values.

This manual also provides an **Index.**

Related Publications

The following publications are referred to in the text of this manual. You may want to use them for more information on certain topics.

* *C Language Manual*, SC09-1128.
* *Interactive System Productivity Facility/Program Development Facility Program Reference for MVS*, SC34-2089.
* *OS/VS2 TSO Terminal User's Guide*, GC28-0645.
* *OS/VS2 MVS JCL*, GC28-0692.
* *OS/VS2 TSO Command Language Reference*, GC28-0646.
* *OS/VS Linkage Editor and Loader*, GC26-3813.
* *MVS/Extended Architecture TSO Extensions TSO Command Language Reference*, SC28-1134.
* *IMS/VS Version 1 Application Programming*, SH20-9026.
* *OS/VS-VM/370 Assembler Programmer's Guide*, GC33-4021.
* *OS/VS2 MVS Utilities*, GC26-3902.

Chapter 1: C Compiler Overview

This chapter explains how the compiler operates. The overview provides a basic understanding of the compiler architecture.

Introduction

The IBM C Compiler for System/370 is an integrated series of software development tools. It reads C source files and generates assembly language, which the assembler translates to object modules. The IBM linkage editor or loader combines object modules to produce executable files. You can request listings that show your C source interspersed with the assembly language that the compiler generates. You can also request that the compiler include a source level interactive debugger with your program.

You begin compilation by invoking the **CC** program. Under MVS, you provide **CC** with the name of your C source file. You can also specify various compiler options. If you plan to use the source level debugger with the program you are compiling, for example, you must specify the **DEBUG** option to **CC**.

When you invoke **CC** and provide it with the appropriate file name and options, the compiler generates an assembly language file, as well as any listings you request. A standard IBM assembler translates the assembly language file to an object module. You then invoke the IBM linkage editor or loader to combine the object modules to make an executable file.

Compiler Architecture

The compiler consists of several programs that work together to translate your C source files to assembly language files and listings. **CC** controls the operation of these programs automatically, using the information you provide in the options you specify to it.

The first step **CC** performs is to run the preprocessor. The preprocessor carries out preprocessor directives and expands macros. **CC** then runs two additional compiler passes to translate your preprocessed source code to an assembly language file. Finally, **CC** runs the IBM assembler you request to convert the assembly language file to an object module.

Linking

Once you have translated all of the separate compilations to object modules, you then run the IBM linkage editor or loader to combine all the object modules that make up your program with the

appropriate members from the C library. You can also build your own libraries and have the linkage editor or loader select files from them as well. The linkage editor or loader generates an executable file which you can invoke. If you specify the debugging option when you invoke **CC**, the linkage editor or loader will select the C library object modules that make up the source level interactive debugger and combine them with your program. When you run your program, any C source compiled with the debugging option will transfer control to the debugger before each executable statement.

Listings

You have several options for obtaining listings. If you request no listings, then **CC** error messages from each of the compiler passes are directed to your terminal or batch output, but no additional information is provided. Each error is labelled with the C source file and line number where the compiler detected the error.

If you request an assembler listing with interspersed C source, the compiler merges the C source as comments among the assembly language statements that it generates. Unless you specify otherwise, the error messages are still written to your terminal as well. **CC** runs a separate utility to produce the combined file which is then used as input to the assembler. Your listing is the listing output from the assembler.

If you request a listing of your C source with no assembly language listing output, **CC** gathers error messages from the three compiler passes and merges them with your C source to produce the listing. Unless you specify otherwise, the error messages are still written to your terminal as well. The same utility for merging the listing is used, but the combined file serves as input to a separate page formatting utility instead of the assembler. Your listing is the output from the page formatting utility.

Chapter 2: Entering C Programs

This chapter tells you how to enter C programs under MVS from within either TSO or ISPF. It also describes the restrictions you need to consider in creating your C source file.

Rules for C Program Source Files

The C language imposes few restrictions on source files. You can place text in the first column of your file, unlike with some other programming languages. You can write C source files that use all of the columns in a text line, if you wish. You can direct the compiler to concatenate a physical text line with the text line following by writing a backslash character at the end of the first physical text line. You must not write a C source file with a text line longer than 511 characters, however, either as a single physical text line or as two or more concatenated lines.

You create C source files using a subset of the EBCDIC character set. The C character set is extensive, so your input device may not be able to produce some of the characters you need to write your C program. The compiler lets you write special "trigraph" sequences to represent some characters. A trigraph is a series of three input characters which the compiler converts to the appropriate member of the C character set. You can replace the left brace, for instance, with the trigraph **??<**. The replacement is permissible *anywhere* in your source code.

For more information on the C character set and on trigraphs, see Chapter 2, "Elements of the C Language," in your *C Language Manual* (SC09-1128).

Entering Programs with the ISPF Editor

You create C source files under ISPF in the same way that you create any other kind of file. Use the ISPF **Edit** option. You must enter your program using both uppercase and lowercase letters using the **Caps Off** option, since case is significant in C. For example, you must write keywords (such as **while**) in lowercase, but the type definition **FILE** must be in uppercase.

The C compiler accepts input in any record format and with any record length up to 511 characters. However, it will ignore columns 73 to 80 of a fixed record format file unless you specify the **NOSEQ** option to **CC**. Specify a value for **lrecl** that best suits your needs. A value of 80 for **lrecl** is probably a good choice most of the time. The last qualifier of the dataset name can be

C.

For more information on ISPF, read *Interactive System Productivity Facility/Program Development Facility Program Reference for MVS* (SC34-2089).

Entering Programs under TSO

You create C source files under TSO in the same way that you create any other kind of file. Use the TSO editor **EDIT**. You must enter your program using both uppercase and lowercase letters, since case is significant in C. For example, you must write keywords (such as **while**) in lowercase, but the type definition **FILE** must be in uppercase.

The C compiler accepts input in any record format and with any record length up to 511 characters. However, it will ignore columns 73 to 80 of a fixed record format file unless you specify the **NOSEQ** option to **CC**. Choose the logical record length that best suits your needs. A value of 80 for **lrecl** is probably a good choice most of the time. You may also wish to change the record format to **V** (variable) in many cases. The descriptive qualifier should be **C**.

For more information on TSO, read *OS/VS2 TSO Terminal User's Guide* (GC28-0645) or *OS/VS2 TSO/Command Language Reference* (GC28-0646).

Chapter 3: Compiling Programs

This chapter tells you how to compile programs in an MVS environment.

The C compiler translates your C source file into an object module in several stages. Under MVS, the compiler uses datasets for source input, object module output, and several intermediate files. To identify these datasets to the compiler, you must associate them with DDnames. Use the TSO **ALLOCATE** command or the JCL **DD** statement to connect DDnames to datasets.

The specific DDnames that you must define are:

DDname	*Use*
SYSIN	source program written in C
INCLUDE	partitioned dataset for **#include** preprocessor directives
SYSPUNCH	assembler code produced by the C compiler
SYSLIN	object module, assembler output
SYSUT1	workfile for the assembler
SYSUT2	workfile for the assembler
SYSUT3	workfile for the assembler
SYSPRINT	output file for listings
SYSTERM	terminal output file

The compiler does not allow concatenation of multiple datasets as input.

Definition of the following DDnames is optional:

DDname	*Use*
SYSUT10	temporary dataset
SYSUT11	temporary dataset
SYSUT12	temporary dataset
SYSUT13	temporary dataset
SYSUT14	temporary dataset
SYSUT15	temporary dataset

If you define any or all of these optional DDnames, the compiler will use them as temporary datasets. You can compile larger programs by providing larger allocations for the datasets that these DDnames define. You may enhance compiler performance by specifying **UNIT = VIO** for these datasets. If you do not define **SYSUT10**, the compiler will dynamically allocate all of its temporary

datasets. Do not provide any DCB information on these data definitions.

If you do not request any options, the compiler will use only default information to compile and assemble your code. You change the operation of the compiler by specifying the options you want when you run it. You use the same options under TSO, ISPF, and from within a batch job. Follow the rules described for the environment you want to use.

The part of the option name that appears in capital letters is the part of the name you *must* type for the compiler to interpret the option correctly. Specify at least that part of the option name. For example, **RE**, **REN**, and **RENT** all invoke the **REnt** option.

Compiler Options

The options are:

(NO)ASM – When you specify **ASM**, the compiler produces an object module on **SYSLIN**. When you specify **NOASM**, the compiler writes the assembler text to **SYSPUNCH**, and does not generate an object module. If you specify **ASM** and the compiler detects an error, it will not invoke the assembler. The default is **ASM**.

CSect(name) – When you specify **CSECT**, the compiler produces a control section (CSECT) with the name *name*. When you specify the **RENT** option in combination with **CSECT**, the compiler also produces a second CSECT with the name *$name*. Therefore, *name* can be eight characters long when you specify **CSECT** alone, or seven characters when you specify **CSECT** in combination with **RENT**. The CSECT name you specify must not conflict with external function names or with external data object names. See Chapter 8, "Interfaces with Other Languages," for information on the conversion of C names to object module symbol names.

(NO)DEBug – When you specify **DEBUG**, the compiler produces function calls that enter the debugger before each executable statement. Do not specify the **RENT** option in combination with **DEBUG**. If you specify **DEBUG** and **RENT** together the compiler generates an error message and halts. Automatic register allocation is disabled when you specify **DEBUG**. The default is **NODEBUG**, which produces no debugger calls.

Define(name=def,name=def, ...) – When you specify this option, the compiler defines one or more macros before it reads any of the source file you specify. Use this option to define symbols when you invoke the compiler instead of within your C source. Defining a symbol in this way is equivalent to writing a preprocessor directive of the form

#define *name def*

in your C source. Each definition has the form *name=def*. If you omit the *=def* part of the directive, the compiler assumes the definition to be "1". You may enter up to

ten definitions, separated by commas. You may not define parameterized macros.

(NO)HASM – When you specify **HASM**, the compiler invokes the Assembler-H. When you specify **NOHASM**, it invokes the Assembler-XF. By default, the compiler uses the IBM assembler specified as the default assembler when the compiler was installed.

(NO)IDENT8 – When you specify **IDENT8**, the compiler restricts the number of significant characters in internal identifiers to eight characters. This option provides support for C programs written for older compilers, where the programs depend upon limited identifier significance. The default is **NOIDENT8**, which restricts the number of significant characters in internal identifiers to 31 characters. Note that the compiler always restricts the length of external identifiers to eight characters for functions and seven characters for data objects.

(NO)LIBH – When you specify **LIBH**, the compiler strips the '**.h**' suffix from the name of **#include** files that it searches for as members of a partitioned dataset. When you specify **NOLIBH**, **#include** file names that end in the suffix '**.h**' will correspond to partitioned dataset members with names ending in the suffix '**$H**'. The default is **NOLIBH**.

LINecount(n) – This option specifies *n* as the number of lines you want the compiler to include in each page of a listing, including heading lines and blank lines. When you request a C source listing only, *n* can be any integer between 5 and 32767. When you request a listing that contains assembly language source, *n* can be any integer between 5 and 99 if you use Assembler-H or between 5 and 999 if you use Assembler-XF. See *OS/VS-VM/370 Assembler Programmer's Guide* (GC33-4021) for information on IBM assemblers. The default is **LINECOUNT(60)**.

(NO)LISt and (NO)SOurce – Listings help you review your code and error listings. You use these two options separately or in combination to obtain the kind of listing you want. When you specify **LIST** alone, the compiler produces an assembly language listing, provided it detects no errors. If the compiler detects an error when you specify **LIST** alone, it produces a listing of C source interspersed with error messages. When you specify **SOURCE** alone, the compiler produces a listing of C source code and error messages. When you specify both **LIST** and **SOURCE**, the compiler produces a listing of C source code interspersed with assembly language code, provided it detects no errors. If the compiler detects an error when you specify both listing options, it produces a C source listing interspersed with error messages. If you specify two or more options which conflict, and are therefore invalid, the compiler generates an error message and halts. The following options are invalid in the combinations listed:

PPONLY and **LIST**
PPONLY and **SOURCE**
LIST and **NOASM**

The defaults are **NOLIST** and **NOSOURCE**, which produce no listings.

LSEarch(name) – This option allows you to specify an alternate location for **#include** files you enclose in double quotation marks. *name* specifies one of the following:

1. The name of a partitioned dataset or
2. A DDname of the form **dd:***name*, where *name* is a DDname defining a partitioned dataset.

When you specify **LSEARCH**, the compiler first searches for the files you include in your program as members of the partitioned dataset or DDname you specify with *name*. The compiler then searches for the files in the same partitioned dataset that it searches for **#include** files you enclose in angle brackets, as described under **SEARCH** below. The default is to search for **#include** files you enclose in double quotation marks as a sequential dataset, and then to search for them in the same partitioned dataset the compiler searches for **#include** files you enclose in angle brackets. For information on how the compiler searches for included files, see Chapter 7, "The Preprocessor," in your *C Language Manual* (SC09-1128).

(NO)MACarg – When you specify **MACARG**, the compiler expands macro argument identifiers within string constants, as the UNIX™ System V C compiler does. The default is **NOMACARG**, which does not expand string constants.

(NO)PPonly – When you specify **PPONLY**, the compiler runs only the preprocessor. This expands all macros and shows all **#include** files in the C source, without compiling the source. The compiler writes the expanded source to **SYSPUNCH**. The default is **NOPPONLY**, which compiles your source to an object module.

(NO)REnt – When you specify **RENT**, the compiler produces reentrant code by making every access to a data object through a base register address. Data objects can therefore be placed at arbitrary locations in memory. You must specify **CSECT** in combination with **RENT**. Do not specify the **DEBUG** option in combination with **RENT**. If you specify **RENT** and **DEBUG** together the compiler generates an error message and halts. The default is **NORENT**, which does not produce reentrant code unless your program contains no global or static data objects. Chapter 4, "Linking and Loading Programs," tells you how to make your program reentrant.

SEarch(name) – This option allows you to specify an alternate location for preprocessor **#include** files that you enclose in angle brackets. *name* specifies one of the following:

UNIX is a registered trademark of AT&T Bell Laboratories

1. The name of a partitioned dataset or
2. a DDname of the form **dd:***name*, where *name* is a DDname defining a partitioned dataset.

When you specify **SEARCH**, the compiler first searches the partitioned dataset or DDname you specify with *name*. The compiler then searches the dataset defined by the DDname **INCLUDE** for the member name enclosed in angle brackets. The **SEARCH** option affects preprocessor **#include** files that you enclose in double quotation marks as discussed under **LSEARCH** above. For more information on how the compiler searches for **#include** files, see Chapter 7, "The Preprocessor" in your *C Language Manual* (SC09-1128).

SEQ(m,n)|NOSEQ – When you specify **SEQ(***m,n***)**, the compiler ignores columns *m* through *n*, inclusive, in each line of your C source. If you specify **NOSEQ** in combination with **SEQ(m,n)**, the compiler generates an error message and halts. If you specify an invalid range of columns to ignore, the compiler generates an error message and halts. The default for variable length record format files is **NOSEQ**, which does not ignore any columns. The default for fixed record format files is **SEQ(73,80)**.

(NO)SHowinc – When you specify **SHOWINC**, the compiler shows the contents of **#include** files in listings. When you specify **NOSHOWINC** the compiler does not show the contents of included files. The default is **NOSHOWINC**.

(NO)TErminal – When you specify **TERMINAL**, the compiler writes error messages to the **SYSTERM** dataset. When you specify **NOTERMINAL**, the compiler writes error message output to **SYSPRINT**. If a listing is being sent to **SYSPRINT** at the same time, the error message error output is lost. The default is **TERMINAL**.

(NO)UPCONV – When you specify **UPCONV**, the compiler uses "unsignedness preserving" rules for type conversions. You use this option to compile older C programs that may depend on the older conversion rules. The default is **NOUPCONV**, which means the compiler uses the "value preserving" rules adopted in the proposed ANSI standard for C.

Invoking the Compiler under TSO

To compile programs using TSO, you must first allocate the files you need for compilation. The **ALLOCATE** command (or the **ALLOCATE** subcommand of the **EDIT** program) dynamically allocates the files the compiler requires. You then use the **CALL** command to run the compiler. You specify the name of the program you want to compile as the definition of the DDname **SYSIN**.

To pass options to the compiler, you enter the **CALL** command, followed by the filename of the compiler (**SYS1.C370.LINKLIB(CC)** in the example that follows), followed by the compiler options you wish to use. Separate options with commas, and surround the

options with single quotation marks.

The following sample CLIST compiles the program **ECHO**. A source code listing of the **ECHO** program appears in Chapter 6, "Debugging Programs." The identification qualifier is **JONES**. The object module output is **JONES.ECHO.OBJ**. In this example, the compiler searches for **#include** files in dataset **XYZ** and uses Assembler-H instead of Assembler-XF. You must modify this sample CLIST to meet the specific requirements of your system. In particular, **SYSDA** may not be the correct unit designation for your installation.

```
ALLOCATE DDNAME(SYSIN) DATASET('JONES.ECHO.C') SHR
ALLOCATE DDNAME(INCLUDE) DATASET('SYS1.C370.MACLIB') SHR
ALLOCATE DDNAME(SYSPUNCH) UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSLIN) DATASET('JONES.ECHO.OBJ')    +
         NEW SPACE(1,1) CYLINDERS UNIT(SYSDA) CATALOG
ALLOCATE DDNAME(SYSUT1) NEW UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSUT2) NEW UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSUT3) NEW UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSTERM) DATASET(*)
ALLOCATE DDNAME(SYSPRINT) DATASET(*)
CALL 'SYS1.C370.LINKLIB(CC)' 'SEARCH(XYZ),HASM'
```

You often need to allocate several files to perform one compilation. For more information on TSO, read *OS/VS2 TSO Terminal User's Guide* (GC28-0645) or *OS/VS2 TSO Command Language Reference* (GC28-0646).

Invoking the Compiler in a Batch Job

To compile in the background, you must write your commands in JCL (Job Control Language). Every job requires the use of the **JOB** statement (to identify the job), **EXEC** statements (to identify each job step), and **DD** statements (to identify datasets the job uses). In the following example, the program **ECHO** is stored in a sequential dataset named **'JONES.ECHO.C'**. The header files are stored in **'SYS1.C370.MACLIB'**, which is a partitioned dataset. The link library **'SYS1.C370.LINKLIB'** is also a partitioned dataset.

To specify compiler options in a batch job, add the option specifications to the **EXEC** statement you use to run the compiler. Follow the specification of the program name **CC** with a comma, the sequence **PARM=**, and the option specification. To specify multiple options, enclose the list in single quotation marks and separate the options with commas. In the example batch file, the compiler searches for **#include** files in dataset **XYZ** and uses Assembler-H instead of Assembler-XF.

The following sample JCL compiles the program **ECHO**. A source code listing of the **ECHO** program appears in Chapter 6, "Debugging Programs." The identification qualifier is **JONES**. The object module output is **ECHO.OBJ**. You must modify this sample JCL to meet the specific requirements of your system. In particular, **SYSDA** may not be the correct unit designation, and the job card in the example may not be correct for your installation.

```
//IBMC  JOB (...),'COMPILE ECHO',REGION=512K
//COMP     EXEC PGM=CC,PARM='SEARCH(XYZ),HASM'
//STEPLIB    DD DSNAME=SYS1.C370.LINKLIB,DISP=SHR
//SYSIN      DD DSNAME=JONES.ECHO.C,DISP=SHR
//INCLUDE    DD DSNAME=SYS1.C370.MACLIB,DISP=SHR
//SYSPUNCH   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLIN     DD DCB=(BLKSIZE=800,LRECL=80,RECFM=FB),
//           DISP=(NEW,CATLG,DELETE),
//           DSNAME=JONES.ECHO.OBJ,
//           SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT1     DD UNIT=SYSDA,SPACE=(CYL,(1,1)),
//SYSUT2     DD UNIT=SYSDA,SPACE=(CYL,(1,1)),
//SYSUT3     DD UNIT=SYSDA,SPACE=(CYL,(1,1)),
//SYSPRINT   DD SYSOUT=*
//SYSTERM    DD SYSOUT=*
```

For more information on JCL, read *OS/VS2 MVS JCL* (GC28-0692).

Invoking the Compiler under ISPF

There is no ISPF compilation window to run the C compiler. To run the C compiler in the foreground, go to the TSO command window and invoke the compiler using TSO commands as described in the section "Invoking the Compiler under TSO" above.

To run the C compiler in the background, edit and submit a file of JCL statements as you ordinarily would in ISPF. Edit a file of JCL statements using the ISPF editor as described in Chapter 2. The JCL statements you need to compile a C program are described in the section "Invoking the Compiler in a Batch Job" above. You can then submit the file of JCL statements as a background job using the TSO **SUBMIT** command or the ISPF editor **SUBMIT** command.

Chapter 4: Linking and Loading Programs

The C compiler produces as output an object module. You cannot invoke this object module directly. You must link the object modules for any program you write with support functions that provide the appropriate runtime environment for program execution. The C library **'SYS1.C370.CLIB'** contains these runtime support functions, as well as the functions described in Chapter 11, "C Library Reference," in the *C Language Manual* (SC09-1128).

Using the Linkage Editor under ISPF

You load object modules from ISPF/PDF using TSO commands or CLISTs. See "Using the Linkage Editor and Loader under TSO" below for a discussion of TSO load commands.

You may use ISPF/PDF to run the linkage editor either in the Foreground or as a Batch processing job. You can send instructions to the linkage editor from both the Foreground and Batch menus. The Foreground menu includes an entry panel that enables you to invoke the linkage editor. Specify the C library as **'SYS1.C370.CLIB'** under the "Additional Input Libraries" entry panel. You can specify up to two libraries to the linkage editor using the Foreground menu. **'SYS1.C370.CLIB'** must be one of these libraries.

ISPF/PDF requires that you supply the full partitioned dataset name of your object modules when you pass them to the linkage editor. It requires the descriptive qualifier **OBJ**.

The linkage editor options field allows you to make several choices about how you want to link edit your object module. You will not need to issue linkage editor control statements for simple program linkage.

The *ISPF/PDF Program Reference* (SC34-2089) contains more information on ISPF/PDF.

Using the Linkage Editor and Loader under TSO

Under TSO, you can link and load your object modules in one of several ways. You run the linkage editor using the command **LINK**. **LINK** allocates the necessary partitioned datasets and links your object modules using the options and library routines you specify. **LINK** then saves the resulting load module in a partitioned dataset with the descriptive qualifier **LOAD**. You can then run the program using the **CALL** command. See Chapter 5, "Executing Programs," for more information on running C

programs.

The form of the **LINK** command is:

```
LINK (<object module name(s)>) LOAD
                    (<load module name>(<membername>)))
```

In this example, you tell **LINK** where you want the linkage editor to place the load module. You can specify the partitioned dataset name for the object module with or without a descriptive qualifier. If you omit the descriptive qualifier, it becomes **OBJ** by default.

You can specify most simple linkage editor options with simple parameters as shown above. You specify more complex options by using linkage editor control statements.

The TSO **LOADGO** command directs the loader to link, edit, and run your object module. Using **LOADGO** is much like to using the **LINK** and **CALL** commands described above, except that it does not save the load module. You can also use **LOADGO** to run existing executable modules. If your application requires any datasets, create or allocate them first using the **ALLOCATE** command.

The **LIB** parameter of the **LINK** or **LOADGO** commands allows you to specify the libraries you want it to search for external references. First enter the **LIB** parameter following your **LINK** or **LOADGO** command. Then enter the names of the library partitioned datasets, in the order you want **LINK** or **LOADGO** to search them. For instance, if you want to load the program **'JONES.ECHO.OBJ'**, and search the library **'LIB17.MGR.NLIB'**, use the following command:

```
LOADGO ('JONES.ECHO.OBJ') 'HELLO, WORLD!'
                LIB('LIB17.MGR.NLIB' 'SYS1.C370.CLIB'))
```

Remember to include the C library **'SYS1.C370.CLIB'**. The *OS/VS2 TSO Terminal User's Guide* (GC28-0645) and *TSO Command Language Reference* (GC28-0646) contain more information about using TSO commands to link and load object modules. A source code listing of the **ECHO** program appears in Chapter 6, "Debugging Programs."

Using the Linkage Editor and Loader in a Batch Job

You use JCL to create batch jobs that load, link, and/or run object modules and load modules. The sample batch file below loads and runs the program **'JONES.ECHO.OBJ'**. Note that the C library **'SYS1.C370.CLIB'** is made available to the loader. You must modify this sample batch file to meet the specific requirements of your system. In particular, the job card in the example may not be correct for your installation.

```
//JONES     JOB (...),'LOAD AND EXECUTE ECHO',REGION=512K
//GO        EXEC PGM=LOADER,PARM=(NOPRINT,TERM)
//SYSLIB    DD DSNAME=SYS1.C370.CLIB,DISP=SHR
//SYSLIN    DD DSNAME=JONES.ECHO.OBJ,DISP=SHR
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(10,10))
//SYSTERM   DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*
//SYSERR    DD SYSOUT=*
```

This job directs the operating system to run the loader in jobstep **GO** to load the program **JONES.ECHO.OBJ** and run it immediately. It is equivalent to the TSO **LOADGO** command. The **EXEC** statement runs the loader. The **PARM** parameter allows you to specify the options you want to pass to the loader. **SYS1.C370.CLIB** is the C runtime library, which you must include when you link or load a C program. The object module name is **JONES.ECHO.OBJ**.

The sample batch file below links the object module **'JONES.ECHO.OBJ'**. Note that the C library, **'SYS1.C370.CLIB'**, is made available to the linker. You must modify this sample batch file to meet the specific requirements of your system. In particular, **SYSDA** may not be the correct unit designation for your installation.

```
//JONES     JOB (...),'MAKE LOAD MODULE ECHO',REGION=512K
//LINK      EXEC PGM=LINKEDIT,PARM=(NOPRINT,TERM)
//SYSLIB    DD DSNAME=SYS1.C370.CLIB,DISP=SHR
//SYSLIN    DD DSNAME=JONES.ECHO.OBJ,DISP=SHR
//SYSLMOD   DD DSNAME=JONES.LOAD(ECHO),
//          DISP=(NEW,CATLG),SPACE=(CYL,(1,1,1)),UNIT=SYSDA
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(10,10))
//SYSTERM   DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*
//SYSERR    DD SYSOUT=*
```

Jobstep **LINK** invokes the linkage editor and produces a load module in a manner similar to the TSO **LINK** command. The sample batch file below runs the load module **'JONES.LOAD'**. The arguments to the program are the strings "HELLO," and "WORLD!". You must modify this sample batch file to meet the specific requirements of your system.

```
//JONES JOB (...),'EXECUTE ECHO',REGION=512K
//ECHO  EXEC PGM=ECHO,PARM=('HELLO, WORLD!')
//STEPLIB   DD DSNAME=JONES.LOAD,DISP=SHR
//SYSPRINT  DD SYSOUT=*
//SYSERR    DD SYSOUT=*
```

The jobstep **ECHO** runs the load module stored in the partitioned dataset **'JONES.LOAD'**.

The *OS/VS2 MVS JCL* (GC28-0692) manual contains more detailed information on using JCL. Read *OS/VS Linkage Editor and Loader* (GC26-3813) for more information on linking and loading programs.

Linking for Execution under IMS

When you write a C program to run under IMS, the function **main** serves as the entry point. The compiler compiles your program to run under IMS in the same way as it compiles a program to run under MVS or MVS/XA.

The following sample batch file links the program **'JONES.IMSAPP.OBJ'** to run under IMS. You must modify this sample batch file to meet the specific requirements of your system.

In particular, the job card in the example may not be correct for your installation.

```
//JONES     JOB (...),'MAKE LOAD MODULE IMSAPP',REGION=512K
//LINK      EXEC PGM=LINKEDIT,PARM=(NOPRINT,TERM)
//SYSLIB    DD DSNAME=SYS1.C370.CLIB,DISP=SHR
//IMSLIB    DD DSNAME=IMSVS.RESLIB,DISP=SHR
//SYSLIN    DD DSNAME=JONES.IMSAPP.OBJ,DISP=SHR
//          DD *
 INCLUDE SYSLIB(CRTI)
 INCLUDE IMSLIB(DFSLI000)
 NAME IMSAPP(R)
/*
//SYSLMOD   DD DSNAME=JONES.IMS.LOAD,DISP=OLD
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSTERM   DD SYSOUT=*
//SYSTERM   DD SYSOUT=*
```

For more information on IMS, see *IMS/VS Version 1 Application Programming* (SH20-9026).

Controlling Stack Size

Your program allocates calling environments and data objects with dynamic lifetime within a storage area called a "stack." By default, the stack occupies 50,000 bytes. If your program terminates abnormally because the signal **SIGSTACK** was reported, it requires a larger stack.

A "stack overflow" is often the result of a program error. If you call a function with uncontrolled recursion, your program will eventually run out of stack. If you declare very large arrays with storage class **auto** or **register**, your program may report a stack overflow when the block that declares these large data objects is entered. Do not increase stack size to overcome a program error.

If your program requires a larger stack, you can change the stack size. To increase stack size (to 80,000 bytes in this example), add a file level definition of the form

```
const int _stksize = 80000;          /*  for 80,000 byte stack  */
```

to any one of the compilations making up your program. Recompile and relink the program, and it will run with the stack size you specify as the initial value of **_stksize**.

Reentrant Programs

Making a program reentrant can significantly decrease system memory usage if the program is frequently run by more than one user simultaneously. When multiple invocations of a reentrant program run, each user has an individual copy of the data objects, but all share a single copy of the program instructions. Do not make a program reentrant that is seldom run by more than one user at a time. A single invocation of a reentrant program requires more memory than an identical program that is not

reentrant.

Creating Reentrant Programs

A C program that does not modify static data objects is reentrant without compiling or linking for reentrancy. Other C programs can be made reentrant. Restrictions concerning reentrant programs are discussed below. You compile a program for reentrancy by specifying the **RENT** and **CSECT(***name***)** options to **CC**. When you specify the **RENT** option, the compiler compiles your program so that data objects can by accessed relative to a base address register. Data objects can therefore be placed at arbitrary memory locations. When you specify the **CSECT(***name***)** option, the compiler creates a control section (CSECT) and external dummy section (DXD) used to hold static data objects. These processes are described below.

After you compile a program for reentrancy you must process it using the **RLINK** program before creating an executable module. **RLINK** processes the external dummy sections the compiler creates and provides initialization data as described below. The output of **RLINK** is a single reentrant object module which you use as input to the linkage editor or loader. Appendix A, "Compile Time Error Messages," in your *C Language Manual* (SC09-1128) discusses the error messages that **RLINK** generates.

On MVS and MVS/XA, you invoke **RLINK** as a batch job, as described below. After you run **RLINK** you run the linkage editor or loader to create an executable module. On MVS and MVS/XA, you must put a reentrant program into the Link Pack area so that multiple users can invoke it. This process is described below.

How the Compiler Processes a Reentrant Program

In reentrant programs, the compiler associates a pair of external identifiers with all non "const" data objects. When compiling the program, the compiler truncates external identifiers to seven characters. To create the first member of the pair, the compiler prepends a '**$**' character to the truncated identifer. This identifier names an external dummy section. To create the second member, the compiler prepends a '**#**' character to the truncated identifier. This identifier names a constant that holds the initial value of the data object. For example, a data object you declare as

```
int maxval = {1000}
```

has associated with it the assembly language statements **$MAXVAL DXD 4C** and **#MAXVAL DC F'1000'**.

The compiler writes static data objects into the external dummy section **$**<*csect*> and into the CSECT **#**<*csect*>, where <*csect*> is the name you provide when you specify the **CSECT(***name***)** option to **CC**. No global function or data object may have the same name as a CSECT name you specify.

How RLINK Processes a Reentrant Program

RLINK takes as input one or more object modules and an optional object module library and produces as output a single object module.

RLINK first searches the object module library you specify to satisfy external references. **RLINK** searches for undefined symbols as members of the partitioned dataset you define with the DDname **INCLUDE**. The DDname **INCLUDE** must define an object module library and not a load module library. Note that **RLINK** will not generate an error message if it encounters references it cannot resolve. These references will remain unresolved in the output object module. The DDnames you must define are described below.

RLINK then collects external dummy sections. It calculates the size and offset of each dummy section, resolves all references to it, and calculates its total size. **RLINK** does not leave any information about external dummy references in the object modules it processes.

Next, **RLINK** creates the control section **C#INIT1**, which contains initialization information that the C runtime uses to initialize the allocated data section. If you do not run **RLINK** to process your program, the linkage editor or loader will use the dummy table **C#INIT1** that the C library defines. Because the C library defines a dummy version of this table when you do not specify the **RENT** option, do not specify the C library as input to **RLINK**. For information on the C runtime, see Chapter 8, "C Runtime Environment," in your *C Language Manual* (SC09-1128).

On MVS and MVS/XA, you must specify DDname **SYSIN** as the input file. You can pass additional input files to **RLINK** as arguments on the **RLINK** command line. For information about file names on MVS and MVS/XA, see the description of the **fopen** function in Chapter 11, "C Library Reference" in your *C Language Manual* (SC09-1128). Note that **RLINK** does not currently support concatenation of datasets. Run the **IEBGENER** utility to concatenate datasets before you run **RLINK**. **RLINK** sends its output to DDname **SYSPUNCH** on MVS and MVS/XA.

Datasets Required By RLINK

You must define the DDnames listed below and associate them with the datasets described when you invoke **RLINK**.

INCLUDE Must be the partitioned dataset that **RLINK** is to search for symbols to satisfy external references. All members of the partitioned dataset must be object modules of fixed record format and a logical record length of 80.

SYSIN Must contain the object module input to **RLINK**. Each object module must be a fixed record format file with a logical record length of 80. You may use any blocking factor.

SYSPUNCH Will contain the object module output of **RLINK**. The record format of this module must be **fixed** (F) or **fixed block standard** (FBS). The dataset must be

organized as a physical sequential dataset, and it must be a diskfile. The default record format is FBS.

SYSPRINT Will contain any error messages that **RLINK** generates. This dataset can have any record format.

SYSERR Will contain any fatal error messages. This dataset can have any record format.

Examples

To compile, prelink, link and run the C program '**JONES.CSOURCE(ECHO)**', using the object module library of reentrant functions in '**JONES.CRENTLIB**', submit the following JCL. This JCL creates the reentrant module '**JONES.LPALIB(ECHO)**'.

```
//JONES       JOB   (...),REGION=512K
//COMP        EXEC  PGM=CC,PARM='RENT,CSECT(ECHO)'
//INCLUDE       DD  DISP=SHR,DSNAME=SYS1.C370.MACLIB
//STEPLIB       DD  DISP=SHR;DSNAME=SYS1.C370.LINKLIB
//SYSIN         DD  DISP=SHR,DSNAME=JONES.CSOURCE(ECHO)
//SYSLIN        DD  DISP=(NEW,PASS),DSNAME=&&OBJ,
//                  SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSPRINT      DD  SYSOUT=*
//SYSPUNCH      DD  UNIT=SYSDA,DISP=(NEW,PASS),SPACE=(CYL,(1,1))
//SYSTERM       DD  SYSOUT=*
//SYSUT1        DD  SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT2        DD  SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSUT3        DD  SPACE=(CYL,(1,1)),UNIT=SYSDA//*
//RL          EXEC  PGM=RLINK,COND=(0,LT,COMP)
//INCLUDE       DD  DISP=SHR,DSNAME=JONES.CRENTLIB
//STEPLIB       DD  DISP=SHR,DSNAME=SYS1.C370.LINKLIB
//SYSIN         DD  DISP=(OLD,DELETE),DSNAME=&&OBJ
//SYSPUNCH      DD  DISP=(NEW,PASS),DSNAME=&&RLOBJ,
//                  SPACE=(CYL,(1,1)),UNIT=SYSDA
//SYSPRINT      DD  SYSOUT=*
//SYSERR        DD  SYSOUT=*//*
//LKED        EXEC  PGM=LINKEDIT,COND=(0,LT,RL),PARM=(REVS,
//                  RENT, NOPRINT, NOTERM)
//SYSLIB        DD  DISP=SHR,DSNAME=SYS1.C370.CLIB
//SYSLIN        DD  DISP=(OLD,DELETE),DSNAME=&&RLOBJ
//SYSLMOD       DD  DISP=(NEW,CATLG),DSNAME=JONES.LPALIB(ECHO),
//                  SPACE=(CYL,(10,10,10)),UNIT=SYSDA
//SYSPRINT      DD  SYSOUT=*
//SYSUT1        DD  SPACE=(CYL,(1,1)),UNIT=SYSDA
```

Using the Link Pack Area under MVS and MVS/XA

MVS and MVS/XA will only take advantage of the fact that a program is reentrant if you place it in the Link Pack Area. After you create the reentrant load module, you must ask your system administrator to take the actions required to place the load module in the Link Pack Area before you run it.

For information on creating reentrant programs on MVS and MVS/XA, see *MVS Overview* (GC28-0984) and *MVS SPL Initialization and Tuning Guide* (GC28-0681).

Restrictions on Making Programs Reentrant

The current version of IBM C for System/370 does not support the initialization of static data objects (data objects with storage class **extern** or **static**) with the address of a non "const" static data object within reentrant programs. In addition, you must supply all the information concerning a data object with storage class **static** the first time you declare the data object. The current version does not support tentative declarations in a reentrant program, even to the restricted extent supported by the ANSI Draft Proposed Standard for C of April, 1985.

Chapter 5: Executing Programs

You can execute C programs interactively under TSO, or in a batch job.

Executing Programs under ISPF

To execute your program under ISPF/PDF, use Option 6 (Command) of the ISPF/PDF menu. You can also use the ISPF/PDF editor to create a file for execution as a batch job. Use the TSO **SUBMIT** command, or the ISPF/PDF editor **SUBMIT** command, to submit the batch job for execution. "Executing Programs under TSO" below explains the appropriate TSO commands.

For more information on Option 6, see *Interactive System Productivity Facility/Program Development Facility Program Reference for MVS* (SC34-2089).

Executing Programs under TSO

Execute load modules with the TSO **CALL** command. Before using the **CALL** command, specify the DDnames your program requires using the **ALLOCATE** command. Chapter 3, "Compiling Programs," lists the DDnames you must define. The load module must be a member of a partitioned dataset. If you omit the descriptive qualifier and member name, TSO assumes them to be **LOAD** and **TEMPNAME**, respectively. To execute **'JONES.LOAD(ECHO)'**, enter:

```
CALL 'JONES.LOAD(ECHO)' 'ABC DEF'
```

Use the **LOADGO** command to execute object modules without producing a permanent relocatable load module. For information on the **LOADGO** command, read Chapter 4, "Using the Linkage Editor and Loader under TSO." A source code listing of **ECHO** appears in Chapter 6, "Debugging Programs."

You are notified by a system message if your program does not terminate normally.

When executing your program under TSO, the three standard text streams are connected to your terminal. **stdout** and **stderr** are open for writing to the terminal and **stdin** is open for reading from the terminal.

On TSO, **argc** has a value equal to the number of command arguments you specify plus the program name, and **argv[0]** points at the null string. In the example above, **argc** has the value 3

and **argv[0]** points at a null string "".

Executing Programs in a Batch Job

The following sample batch job runs the program **'JONES.LOAD(ECHO)'**. The arguments to **ECHO** are **ABC** and **DEF**. You must modify this sample batch file to meet the specific requirements of your system.

```
//JONES     JOB (...),'EXECUTE ECHO',REGION=512K
//ECHO      EXEC PGM=ECHO,PARM=('ABC DEF')
//STEPLIB    DD DSNAME=JONES.LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSERR    DD SYSOUT=*
```

When executing as a batch jobstep, the three standard text streams are given DDname assignments. The standard input stream **stdin** reads from the file identified by the DDname **SYSIN**. The standard output stream **stdout** writes to the file identified by the DDname **SYSPRINT**. The standard error stream **stderr** writes to the file identified by the DDname **SYSERR**. The files connected to the standard text streams are not actually opened until they are used. Therefore, it is not always necessary to define all three DDnames. However, you should always define **SYSERR** so that runtime error messages will be reported.

On MVS and MVS/XA, **argc** has a value equal to the number of command arguments you specify plus the program name, and **argv[0]** points at the null string. In the example above, **argc** has the value 3 and **argv[0]** points at a null string "".

Read *OS/VS2 MVS JCL* (GC28-0692) for more information on JCL. For more information on redirecting input to and output from your program, specifying multiple command line arguments, and other aspects of command line processing, see your *C Language Manual* (SC09-1128).

Chapter 6: Debugging Programs

The C source level debugger provides a controlled environment for program execution and examination. You invoke the debugger by running an executable file that you compile with the **DEBUG** option. The linkage editor or loader links the debugger components with your program.

Once you enter the debugger (by executing a statement that was compiled with debugging enabled), you can request a variety of services. You can display a particular data object when its stored value changes. You can stop execution when the debugger reaches a particular C source line. You can display each source line of your program before it executes. You can answer questions about the values stored in data objects, or send a copy of your debugger session to a file.

The following examples demonstrate the debugging of the program **'JONES.ECHO.C'**. To follow along with this manual, enter the C source code for **ECHO** program and compile **ECHO** with debugging enabled. The examples step you through the control commands you most often use in a debugging session. For these examples, use TSO commands to enable the debugger.

The source code for the example program **ECHO** is:

```
1   /*  ECHO ARGUMENTS TO STDOUT
2    */
3   #include <stdio.h>
4
5   int main(ac, av)
6       int  ac;
7       char *av[];
8       {
9       ++av, --ac;
10      for (; 0 < ac; ++av, --ac)
11          {
12          printf("%s", *av);
13          if (1 < ac)
14              printf(" ");
15          else
16              printf("\n");
17          }
18      exit(0);
19      }
```

The following sample TSO commands compile **'JONES.ECHO.C'**. You will have to modify these sample commands to meet the specific requirements of your system.

```
ALLOCATE DDNAME(SYSIN) DATASET('JONES.ECHO.C') SHR
ALLOCATE DDNAME(INCLUDE) DATASET('SYS1.C370.MACLIB') SHR
ALLOCATE DDNAME(SYSPUNCH) UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSLIN) DATASET('JONES.ECHO.OBJ') NEW    +
   SPACE(1,1) CYLINDERS VOLUME(PUB001) UNIT(SYSDA) CATALOG
ALLOCATE DDNAME(SYSUT1) UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSUT2) UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSUT3) UNIT(SYSDA) SPACE(1,1) CYLINDERS
ALLOCATE DDNAME(SYSTERM) DATASET(*)
ALLOCATE DDNAME(SYSPRINT) DATASET(*)
CALL 'SYS1.C370.LINKLIB(CC)' 'DEBUG'
```

The compiled program is **'JONES.ECHO.OBJ'**. The DDname **SYSIN** must be allocated to your debugger input commands to execute the debugger in batch. The DDname **SOURCE** must be allocated to your source dataset to execute the debugger in TSO or batch when the source dataset is uncataloged. You can only debug one source dataset at a time if the dataset is uncataloged. The sample program runs without any errors.

To run **ECHO**, enter:

```
LOADGO ('JONES.ECHO.OBJ')'abc def' LIB('SYS1.C370.CLIB')
```

The debugger will display

```
| main():5    |int main(ac, av)
```

on the screen. The first part of the output line, between the vertical bars, tells you the name and source line number of the function in which you are currently executing. The second part, after the second vertical bar, is the contents of the source line.

The symbol **:>** that appears on your screen is the debugger prompt. When you see the **:>** prompt, it means that the debugger is awaiting a command. Type debugger commands only in response to the **:>** prompt.

When you use the debugger, you enter one of two types of commands:

1. a command that controls the flow of execution
2. a command that examines or modifies a location.

All commands are in the following format:

<location> <cmd letter> <args>

<location> specifies an optional location within the program, a data object, or an unsigned decimal integer. *<cmd letter>* specifies a debugger command. *<args>* specifies an optional argument to the command. If you specify a location, you must separate it from the command itself by spaces or horizontal tabs. The debugger interprets *<location>* and *<args>* differently, depending on the specifics of a given command.

Run the Program to Termination

The simplest debugger action is to run the entire program to termination. Type g in response to the :> prompt. The **ECHO** program then generates the output:

```
abc def
```

The **g** command tells the debugger to continue to the next "breakpoint." A breakpoint is a stopping place that you specify with a debugger command, as explained below. Since there are no breakpoints in this example, the program runs to termination.

Stepping One Source Line at a Time

A second option is to run the program by "single stepping" through the entire run one source line at a time. To rerun **'JONES.ECHO.OBJ'**, enter:

```
LOADGO ('JONES.ECHO.OBJ') 'ABC DEF' LIB('SYS1.C370.CLIB')
```

The debugger will display:

```
| main():5    |int main(ac, av)
```

This time, press the <ENTER> key once after each output line the debugger displays. The result is:

```
| main():9    |         ++av, --ac;
| main():10   |         for (; 0 < ac; ++av, --ac)
| main():12   |             printf("%s", *av);
| main():13   |         if (1 < ac)
| main():14   |             printf(" ");
| main():10   |         for (; 0 < ac; ++av, --ac)
| main():12   |             printf("%s", *av);
| main():13   |             if (1 < ac)
| main():15   |             else
| main():16   |               printf("\n");
abc def
| main():10   |         for (; 0 < ac; ++av, --ac)
| main():18   |       exit(0);
```

Note that the debugger displays only those source lines that are executed. It does not display preprocessor directives, function definitions, or other declarations. Also note the output line `abc def` after the execution of line 16. This is the *actual* output of the **ECHO** program. The vertical bars help you distinguish regular program output from debugger messages.

Stepping Through Multiple Source Lines

Normally, pressing <ENTER> steps through the program one source line at a time. You can step more than one source line by specifying a repeat count in front of the **s** command. For example, the debugger command **10 s** steps through your program ten source lines at a time. To try this feature, run **ECHO** and type 500s in response to the :> prompt.

If you instruct the debugger to step through more source lines than there are lines to execute, the program runs to termination.

If you do not want to see the output of the debugger as it steps through the source lines, write a repeat count in front of the **g** command. Rerun **ECHO** and type 5 g in response to the **:>** prompt. The debugger will display:

```
| main():5    |int main(ac, av)
| main():5    |       printf(" ");
```

Exit Debugging

To end a debugging session without stepping through to program termination, simply enter q in response to the **:>** prompt.

Examining Data Objects

Another option available to you through the source level debugger is to observe what happens to the data objects in your program during program execution. As an example, rerun **'JONES.ECHO.OBJ'** again by entering:

```
LOADGO ('JONES.ECHO.OBJ') 'ABC DEF' LIB('SYS1.C370.CLIB')
```

The program will display:

```
| main():5    |int main(ac, av)
```

Type v in response to the **:>** prompt. The following list appears:

```
variables in main()
argument signed char **av
argument signed int   ac
```

If there are global data objects in your program, the debugger lists them under the heading variables in echo. This list includes all the data objects visible to the current function, as defined by the scope and visibility rules of C. Note that the debugger then prompts you for input again. It stops executing source lines and waits for further instructions. Unless you type g or press only <ENTER>, it will continue to wait for additional instructions.

You can also ask the debugger to print the values stored in data objects. In this example, the simplest data object near the top of the list is **ac**. It is the count of the number of command options you supply to **ECHO**. Type :ac p in response to the **:>** prompt to access and print the stored value. The space after the data object name indicates the end of the data object name and the beginning of the "print" command.

```
argument signed int   ac       value: 3
```

There are three command options to **ECHO**: ECHO (the program name), abc, and def. The name of the program always counts as a command option. In this example, watch how **ac** changes. Press <ENTER> two times to execute the next two source lines. The debugger will display:

```
|main():10      |    for (; 0 < ac; ++av, --ac)
|main():12      |        printf("%s", *av);
```

The program adjusts the value stored in the data object **ac** to track the number of command options that have yet to be

processed. Type :ac p in response to the :> prompt to print out the value stored in the data object **ac**. The debugger will display:

```
argument signed int    ac        value: 2
```

To print **av**, of type *pointer to pointer to char*, type :av p in response to the :> prompt. The debugger will display:

```
argument signed char   **av      value: 31274
```

Note: The actual values for **av** *and* ***av** *depend on where the program is loaded into memory.*

You can specify two levels of indirection to print the character pointed to. Type :**av p%d in response to the :> prompt. The debugger will display:

```
argument signed char   **av      value: 31274->31328->129
```

You can display this character as an EBCDIC character. Type :**av p%c in response to the :> prompt to print out the character. The debugger will display:

```
argument signed char   **av      value: 31274->31328->a
```

You can also print out the *pointer to char*. Type :*av p in response to the :> prompt. The debugger will display:

```
argument signed char   **av      value: 31274->31328
```

You can even use a *pointer to char* to print out a string. Type :*av p%s in response to the :> prompt and the value of the second command line option is displayed.

```
argument signed char   **av      value: 31274->abc
```

Displaying Current Source Line and File Name

You can display the current source line and file name during the debugging session. Type a period . in response to the :> prompt to view the current source line. Press <ENTER> to go to the next source line and then type f in response to the :> prompt to display the current file name.

Assuming that you are at line "main( ):10" when you issue the command, the debugger will display:

```
     |main():10    | for (; 0 < ac; ++av, --ac)
file: JONES.ECHO.C
```

The file name displayed is that of the original C source file **JONES.ECHO.C**.

Executing Multiple Debugger Commands

The debugger allows you to string several commands together. Single step to source line 10 and type :ac p;:*av p%s in response to the :> prompt to display the number of options and the contents of the current option.

```
 |main():10    | for (; 0 < ac; ++av, --ac)
argument signed int    ac       value: 1
argument signed char   av       value: def
```

Note that the debugger breaks up multiple commands on an input line and repeats the `:*av p%s` part on an output line by itself.

To specify multiple commands on a single input line (in response to a single `:>` prompt), separate the commands with semicolons. However, certain commands take the remainder of the input line following the one character command name as an argument, regardless of what actually is present on the command line. You cannot follow such a command with other commands. The breakpoint command **b**, for example, interprets the remainder of the input line as actions to be taken if control is transferred to the specified location. The **f** command also ignores the remainder of the input line. The **w**, **>**, and **<** commands interpret the remainder of the input line as a file name. Do not specify other debugger commands in conjunction with any of these commands.

Logging Debugger Output

You can write a file containing a log of all debugger output. For example, load and run **ECHO** and type `w log.1` in response to the `:>` prompt. Here, **w** is the command to write debugger output to a file, and **log.1** is the name of the file where the debugger stores the output. Both parts of the command are necessary to create a log file. Single step down a few source lines, and then type `w` in response to the `:>` prompt. Move down a few more source lines and type `q` in response to the `:>` prompt. Display the contents of the file **log.1** on your terminal screen as you would the contents of any other file on your system. Note that it shows only debugger input and output. The debugger does not copy ordinary output from **ECHO** to the log file.

Locating Source Code in Other Datasets

The **f** command tells you the file name the debugger must use to look up source lines. Follow the **f** command with a space and the fully qualified name of the source file so that the debugger can locate it.

Setting Breakpoints

The **b** command sets or displays "breakpoints." A breakpoint is a condition that you define. You can set a breakpoint for when the value stored in a data object changes, or for when control passes to a particular source line. To see how you use breakpoints, type `:ac b` in response to the `:>` prompt. This command sets a breakpoint for changes in the value stored in **ac**. The debugger will display:

```
set at argument  signed int  ac  []
```

The response [] shows that you did not request any specific action. The default action upon reaching a breakpoint is to wait for further debugger commands from the keyboard. Note that if you wish to step through source lines when the debugger reaches a breakpoint, you must reenter the **s** command. Type **g** in response to the `:>` prompt. Execution resumes and continues until the program reaches the source line "main( ):10." The debugger will

display:

```
 at 31268 argument  signed int   ac modified
|main():10      |   for (; 0 < ac; ++av, --ac)
```

To set a breakpoint at source line 12 in the function, type `main():12 b` in response to the **:>** prompt. The debugger will display:

```
set at main():12          [ ]
```

To list all the breakpoints currently set, type **b** in response to the **:>** prompt. The debugger will display:

```
main():12          [ ]
argument  signed int ac  [ ]
```

One use for setting breakpoints is to help you track what is happening to a given data object. To see how to track a data object, type **g** in response to the **:>** prompt to execute the program until you encounter a step where the program modifies the value stored in the data object **ac**. The debugger will display:

```
| at 31268 argument  signed int  ac modified
    |main():10      |   for (; 0 < ac; ++av, --ac)
```

You can then continue execution until you encounter another source line that alters the value stored in the data object **ac**.

Deleting Breakpoints

The **d** command deletes specified breakpoints. If you associate several breakpoints with the same data object, the **d** command will delete the first one set. To delete other breakpoints associated with the same data object, repeat the same **d** command as you used to delete the first breakpoint set. This will delete one breakpoint at a time beginning with those you set first. The last breakpoint remaining will be the one set most recently. Type `:ac d` in response to the **:>** prompt to delete the breakpoint associated with **ac**. The debugger will display:

```
reset argument  int  ac  [ ]
```

More Debugger Commands

The following is a list of additional source level debugger commands that you can use. To try each one, perform the steps listed beneath its description.

Moving in Stack Frames

There is one additional command you can use to change the scope you are inspecting with the debugger. Write **m** followed by a direction option to move up or down "stack frames" in the direction you specify. Stack frames are the regions of storage that the compiler allocates and deallocates from the region of storage known as the "stack." A stack frame holds the calling environment of the expression that called the function, the argument data objects passed on the function call, and all of the data objects declared within the function that have dynamic lifetime.

You direct movement of the scope with a **u** to move up one stack frame, **d** to move down one stack frame, **t** to move to the top stack frame, and **b** to move to the bottom stack frame. The function **main** is at the top of the stack. You can precede your **m** command with a location specifier. If you specify a location, the debugger moves as many times as necessary to get to a stack frame that is in scope for that location.

Other Examine/Modify Commands

The **t** command displays a complete list of known stack frames from the current stack frame to the top stack frame (usually **main()**). You can specify the option **v** to find the data object names and associated type information for each frame. If you type tv&, the debugger also prints the address of each data object.

The **u** command updates a data object you specify by storing a new value in it. If you do not specify a value, the debugger prompts you for one. The debugger interprets this value according to the format specification you used most recently to print the value stored in the data object. You can use an = in place of a **u**. Both commands have the same effect. You can only apply these commands to location specifiers.

The **v** command lists the name and type information for all data objects the debugger knows of in the current stack frame. If you specify **&**, the debugger prints the address of each data object.

A **w** command writes a log of all debugger input and output to the file name you specify. Make sure a file by the same name as the one you specify does not exist. The log begins immediately. If you do not specify a file, the debugger stops writing to any previous log file.

You can redirect debugger input as well as output. Use a >, followed by a file name, to redirect debugger command output and prompts to the named file. Use a <, followed by a file name, to redirect debugger command input to come from the named file.

Formatting Output from the Debugger

You can specify formats for commands that examine locations within the program. The formats that you can specify are identical to those available to the C library routine **fprintf**. See Chapter 11, "C Library Reference," in the *C Language Manual* (SC09-1128) for a complete description of **printf** format specifiers. A summary of how to format debugger output appears in Appendix B, "Debugger Command Summary" at the back of this manual.

The debugger does not check storage alignment, so printing a *char* data object as a *short* on an invalid boundary may cause unpredictable behavior. In the debugger, a format specification stops at the first semicolon or at the end of the input line.

Here are some examples of the debugger format specifiers and what they do:

`:x p%c`	print *x* as a *char* data object in EBCDIC
`:d p%17.5f`	print *d* as a *double* to 5 places
`:ptr p0x%08X`	print *ptr* as an *int* in hexadecimal with leading 0x
`:x p%o;:x u025`	set the format for *x* to octal and put a newline in *x*

Printing the Contents of Data Objects

The debugger can access data objects both directly and by indirection on a pointer, as in the C language itself. You specify indirection on a pointer data object by writing:

:*<*location*>

<*location*> designates a pointer data object of the appropriate type. For example, the command

```
:*pi p
```

will print the contents of the data object pointed at by the pointer data object **pi**. The type of the data object is *char*, unless you specify otherwise.

You can specify indirection on a pointer data object only as many times as you specify the pointer attribute in your original declaration. If you request too many levels of indirection, the debugger ignores the extra requests. If you write an integer constant to designate a data object at some absolute memory address, you can specify indirection any number of times. Indirection is valid only in conjunction with location specifiers for the **p** and **u** commands. The rest of the command set ignores indirection.

You specify double indirection on the data object **z** by writing:

```
:**z p
```

Type Casts

You write type casts to make the debugger treat a data object as if it had a different type from its declaration. Unlike in C, where a type cast tells the compiler to convert the value of the data object to the specified type, the debugger type cast notation behaves as though the conversion had already taken place. Type casting, like indirection, works only with location specifiers for the **p** and **u** commands. Type casts take the form:

: (<*data_type*>**)***data_object_name* **p**

or

: (<*data_type*>**)***data_object_name* **u**

where acceptable types for <*data_type*> are:

[signed] [unsigned] char
[signed] [unsigned] short
[signed] [unsigned] int
[signed] [unsigned] long
float
double
long double

and *data_object_name* is a location specifier of the second kind. Specify at most two type names in a type cast. For example, to type cast **z** (a *float* in this example) to a *long*, and print the result in hexadecimal, write:

```
:(long)z p%lX
```

You can combine type casts and indirection. To print the contents of the data object pointed at by the data object **pi** as a *long*, write:

```
:*(long)pi p
```

You cannot print or update arrays directly, and you cannot subscript them to obtain a particular element.

You can print the contents of multiple adjacent memory locations, like arrays and structs, but only if the individual elements of the aggregate type are aligned on integer boundaries. Elements of type *int*, *long*, and *float* will always meet this criterion, but problems will result if elements of the structure are of type *char*, *short*, or *double*. The debugger may skip over some members as it advances to integer boundaries.

The debugger prints the warning **(no format)** when you ask it to print an aggregate data object type. This means that there is no default format associated with the data object.

Summary of Location Specifiers

There are three kinds of location specifiers:

1. For designating one or more executable source lines
2. For designating a data object
3. For providing an unsigned decimal integer.

There are three commands which allow location specifiers of the first kind: **b**, **d**, and **m**. Four commands allow location specifiers of the second kind: **b**, **d**, **p**, and **u**. Three commands allow location specifiers of the third kind: **g**, **s**, and **f**. With an **s** or **g** command, the location specifier specifies the number of source lines to process. For the **f** command it specifies the number of columns per horizontal tab stop.

A location specifier of the first kind can take the following forms:

file_name:line_number	(any executable line)
file_name:	(any executable line throughout file)
function():line_number	(current file assumed)
function():	(throughout function named)
:line_number	(current function or file assumed)

(the current line)

A location specifier of the second kind can take the following forms:

file_name:data_object_name
(data object with external linkage)

function():data_object_name
(in scope of function named)

:data_object_name
(in scope of current function)

number
(hexadecimal, decimal, or octal absolute address)

A data object name is an identifier currently in scope as a data object. You can change the current scope with the **m** command. You can write any number of asterisks (indicating indirection) and/or a type cast in front of a data object name.

Chapter 7: Maintaining Libraries

This chapter tells you how to create libraries of load modules, and how to add, delete, replace, and list the members of your libraries.

Creating Libraries

To create a load module library, you must allocate a partitioned dataset. Use the **ALLOCATE** command, as in the example below, and give your library a descriptive qualifier of **'.LOAD'**.

```
ALLOCATE DATASET('JONES.NEWLIB.LOAD') NEW    +
         SPACE(1,1) CYLINDERS DIR(1) CATALOG
LINK ('JONES.SUBA.OBJ') LOAD('JONES.NEWLIB.LOAD(SUBA)')
LINK ('JONES.SUBB.OBJ') LOAD('JONES.NEWLIB.LOAD(SUBB)')
```

The following sample batch file creates the library **'JONES.NEWLIB.LOAD'** with members **'SUBA'** and **'SUBB'**. You must modify this sample batch file to meet the specific requirements of your system.

```
//JONES     JOB (...),'MAKE LOAD MODULE LIBRARY',REGION=512K
//LINK      EXEC PGM=LINKEDIT,PARM=(NOPRINT,TERM)
//SYSLIN    DD DSNAME=JONES.SUBA.OBJ,DISP=SHR
//          DD *
 NAME SUBA(R)
//          DD DSNAME=JONES.SUBB.OBJ,DISP=SHR
//          DD *
 NAME SUBB(R)
//SYSLMOD   DD DSNAME=JONES.NEWLIB.LOAD,DISP=(NEW,CATLG),
//             SPACE=(TRK,(1,1,1)),UNIT=SYSDA
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(10,10))
//SYSTERM   DD SYSOUT=*
```

In this example, a **NAME** statement is terminated with an **(R)**. If you terminate a **NAME** statement with an **(R)**, the linkage editor replaces any preexisting member with the name specified in the **NAME** statement.

Adding a Member to a Library

To add a member to an existing library, use the linkage editor to link edit your object module and place it in the load module library. By convention, you put object modules in datasets with a **'.OBJ'** descriptive qualifier. Issue the appropriate command to name the entry point of the new member.

The following sample batch file adds the member **'SUBC'** to the library **'JONES.NEWLIB.LOAD'**. You must modify this sample batch file to meet the specific requirements of your system.

```
//JONES    JOB (...),'ADD TO LOAD MODULE LIBRARY',REGION=512K
//LINK     EXEC PGM=LINKEDIT,PARM=(NOPRINT,TERM)
//SYSLIN   DD DSNAME=JONES.SUBC.OBJ,DISP=SHR
//         DD *
 NAME SUBC(R)
//SYSLMOD   DD DSNAME=JONES.NEWLIB.LOAD,DISP=OLD
//SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(10,10))
//SYSTERM   DD SYSOUT=A
```

This jobstep adds a new member to an old library, creating a new library.

Use the **LINK** command, as in the example below, to link the new library. The asterisk '*', used as a dataset name, indicates that you will enter control statements from your terminal. The system will prompt you to enter the control statements. A null line indicates the end of your control statements.

```
ALLOCATE DATASET('JONES.NEWLIB.LOAD') NEW
         SPACE(1,1) CYLINDERS DIR(1) CATALOG
LINK ('JONES.SUBA.OBJ',*,'JONES.SUBB.OBJ',*)+
        LOAD('JONES.NEWLIB.LOAD')

IKJJ76080A ENTER CONTROL STATEMENTS
    NAME SUBA(R)
    (null line)
IKJ761111 END OF CONTROL STATEMENTS

IKJJ76080A ENTER CONTROL STATEMENTS
    NAME SUBB(R)
    (null line)
IKJ761111 END OF CONTROL STATEMENTS
```

Under ISPF, use Option 3.2 (Data Set Utility) to allocate a new dataset. Then use the Foreground (Option 4) or Background (Option 5) command menus to link edit your library with the new member. Be sure to include the **NAME** control statements so that the linkage editor or loader can name your library members. For information on the mapping of C identifiers to external names, see Chapter 8, "Interfaces with Other Languages."

Deleting a Member of a Library

Under ISPF/PDF, delete library members with Option 3.2 (Library Utility). Select "D" in the Library Utility menu to delete the library member.

For information on deleting library members in a batch job, read *OS/VS2 MVS JCL* (GC28-0692).

Under TSO, use the **DELETE** command to delete library members, as in the example below.

```
DELETE 'JONES.NEWLIB.LOAD(SUBC)'
```

Replacing a Member of a Library

The procedure for replacing a library member is the same as the procedure for adding a library member. Remember to append the letter 'R' enclosed in parentheses to the member name so that the linkage editor or loader will perform the replacement operation. For an example of replacing a library member, see "Adding a Library Member" above.

Listing Members of a Library

To display a list of library members in ISPF/PDF, use Option 3.1 (Library Utility). Select the blank space on the Library Utility menu to display all members of a library.

For more information on Library Utility and other ISPF/PDF commands, read *Interactive System Productivity Facility/ Program Development Reference for MVS* (SC34-2089).

Use the TSO command **LISTDS** with the **MEMBERS** operand to list the members of a library. Read *OS/VS2 TSO Command Language Reference* (GC28-0645) for complete information on the **LISTDS** command.

To list library members in a batch job, refer to *OS/VS2 MVS Utilities* (GC26-3902).

Chapter 8: Interfaces with Other Languages

The C compiler supports calls to IBM assembly language routines from within a C program. Once called from within a C program, an assembly language routine can access C data objects or call C functions. You can also call functions written using the standard OS calling sequence from C.

Calling Assembly Language Functions from C

You call an assembly language function from C the same way you call a C function. The assembly language function must match the C conventions for:

* Mapping C identifiers to external names
* Transferring control to another function
* Preserving registers
* Passing argument data objects
* Using the stack
* Returning a value
* Returning control to the calling function

This section addresses these issues in the order given.

Mapping C Identifiers to External Names

When you declare a function with external linkage, its external name is formed from the first eight characters of the identifier. All letters are forced to a single case, and all underscores are replaced with @ characters. For example:

function identifier	*external name*
abc	**ABC**
ABC	**ABC**
get_more	**GET@MORE**
hypotenuse	**HYPOTENU**

The entry point of your assembly language function must match the external name that C uses.

When you declare a data object with external linkage, its external name is formed from the first seven characters of the identifier. All letters are forced to a single case, and all underscores are replaced with @ characters. A $ character is prepended to each

name. For example:

data object identifier	*external name*
abc	**$ABC**
ABC	**$ABC**
get_more	**$GET@MOR**
Hypotenuse	**$HYPOTEN**

The representation of data objects in C is described in your *C Language Manual* (SC09-1128) in Chapter 4, "Declarations."

Transferring Control to Another Function

A C function calls another function by placing its address in register 15 (R15) and executing the instruction:

```
BALR 14,15
```

The address of the called function is in R15. The return address is in R14.

Your assembly language function must preserve the contents of R14 to return control to the C function that calls it. It may make use of R15 to address instructions and data within the assembly language function. If there are arguments to the function, your function must use R10 to locate them, as described below.

Preserving Registers

A C function expects that nearly all registers have the same content upon return from the called function that they have when the function is first called. The only exceptions are:

R12 – which is used to return values of integer or pointer type

R14 – which may be in any state

R15 – which may be in any state

F0 – which is used to return values of floating type

F2 – which may be in any state

F4 – which may be in any state

F6 – which may be in any state.

Your assembly language function can make use of a "save area" to preserve all registers that it might modify. The save area consists of 18 words, each 4 bytes long, on a fullword storage boundary. The calling function assures that R13 points to the save area when your assembly language function is entered. By convention, the word at 4(R13) contains the address of a previous save area. The C source level debugger assumes, for example, that all save areas are linked together in this fashion. The debugger also assumes that registers are stored on function entry with the instruction:

```
STM 14,11,12(13)
```

You must follow these conventions in your assembly language function if you want it to function properly with the C source level debugger.

Passing Argument Data Objects

A C function stores all argument data objects in an area pointed at by R10. The area begins with another 18 word save area, on a fullword storage boundary. The calling function sets aside this area entirely for the convenience of the called function. Your assembly language function need not use it. You can use this new save area if your assembly language function must call yet another function. A useful sequence of instructions at function entry is:

```
STM  14,11,12(13)
ST   13,4(10)
LR   13,10
```

which saves the registers as described above, links the new save area to the older one, and makes the new save area the current one for the function. Your assembly language function is now ready to call another function and provide it with a save area pointed at by R13.

Argument data objects are just above this new save area. If the function returns a structure type, the arguments begin at 76(R10) and continue to higher addresses. For all other functions, the arguments begin at 72(R10). All values of pointer type occupy four bytes. All values of integer type are converted by the calling function to a four byte integer representation. All values of floating type are converted by the calling function to an eight byte *double* representation, beginning at an offset that is a multiple of eight bytes. Therefore, if you call a function with a mixture of argument types, there may be four byte "holes" before some of the arguments of floating type.

All arguments of structure type begin at an offset that is a multiple of four bytes, if the structure contains no component of type *double* or *long double*. An argument of structure type begins at an offset that is a multiple of eight bytes if the structure contains any component of type *double* or *long double*. If you call a function with a mixture of argument types, therefore, there may be four byte holes before some of the arguments of structure type. There may also be one to seven byte holes *after* arguments of structure type. For example:

```
struct {
     char c1, c2;
     } cs;
int i, j;
double d;

j(cs, i, j, d);
```

will place

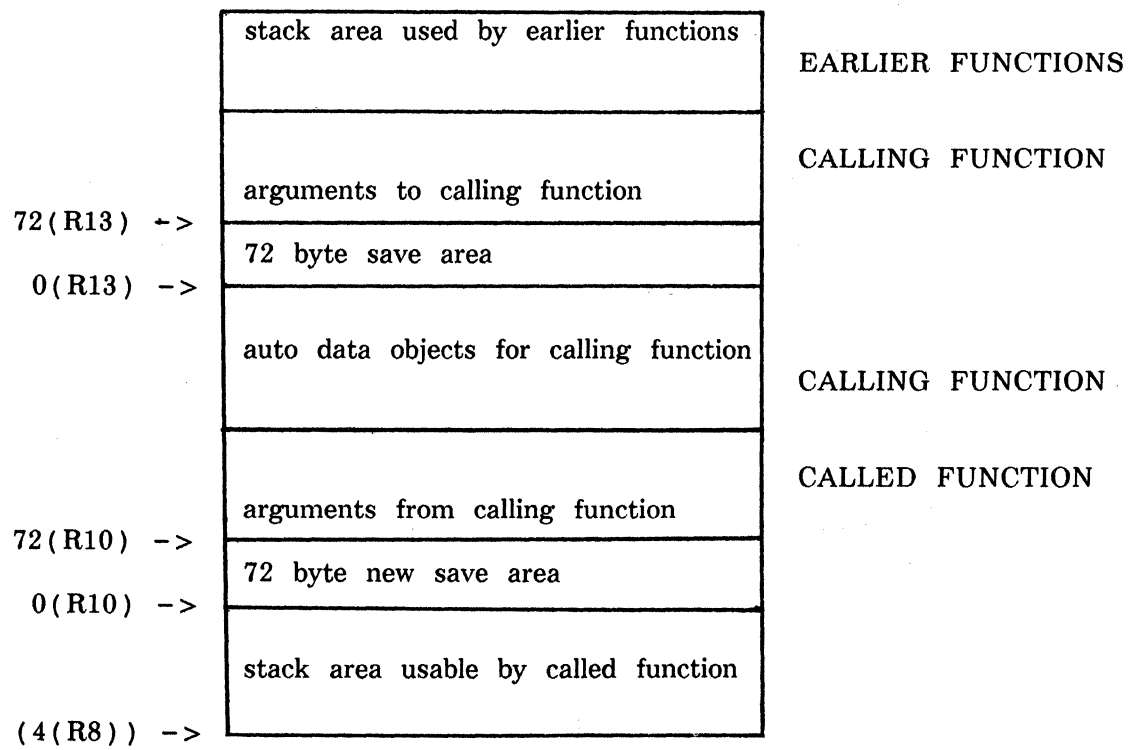
```
cs at 72(R10) occupying 2 bytes
i at 76(R10) occupying 4 bytes
j at 80(R10) occupying 4 bytes
d at 88(R10) occupying 8 bytes
```

There are holes after **cs** and **j**.

Your assembly language program cannot determine how many arguments are present, or what their types are. It may alter the value stored in any of the arguments, since all are deallocated upon return.

Using the Stack

The save areas and argument data objects described above all reside within a storage area called a "stack." A C function allocates calling environments and data objects with dynamic lifetime within the stack, using a "last in first out" discipline. When your assembly language function is entered, the stack looks like:



The area beginning at 72(R13), "arguments to calling function," is shown here to illustrate how the pattern of stack usage repeats for each function call. Your called function should never access any of these argument data objects directly, since the calling function assumes that its argument data objects are for its private use.

You can use for temporary storage the area immediately below 0(R10), as long as you do not access storage below the address

stored at 4(R8). If you call a C function from your assembly language function, however, you must allocate any storage you use in this area. You allocate storage on the stack by subtracting from R10 the number of bytes of storage you need, rounded up to a multiple of eight bytes. Remember to set aside the storage beginning at 0(R10) for the save area and argument data objects you need to call any C functions. The storage allocated for the private use of your assembly language function begins above the last argument data object for any C functions that you call.

The section "Controlling Stack Size" in Chapter 4, "Linking and Loading Programs," tells you how to alter the size of the stack area.

Returning a Value

A function returning any integer or pointer type returns with the value in R12. You must convert all integer values to a four byte representation. A function returning any floating type returns with the value in F0. You must convert all floating values to type *double*, which occupies eight bytes.

A function returning a structure type assumes that the four byte area at 72(R10) contains a pointer to a data object of the structure type. The assembly language function must store a return value in this data object before it returns. The value stored in the data object pointed at by 72(R10) is undefined when the function is first entered. There is no way to determine the size of the structure data object within your assembly language function.

A function returning *void* returns no value.

Returning Control to the Calling Function

To return from a function, compute and store the return value, restore any registers that must be preserved, and branch to the address that was in R14 when your assembly language function was first entered. If you used the entry sequence described earlier, you can return with the exit sequence:

```
L    13,4(0,13)
LM   14,11,12(13)
BR   14
```

Any code sequence that returns to the same location with the necessary registers restored is also acceptable.

Calling Functions Written in Other Languages from C

You use the OS calling sequence to call functions written in languages other than C from within a C program. You call a function that uses the OS calling sequence by declaring the function as you normally would, except that in addition you must write the preprocessor directive

#pragma linkage (*func***, OS)**

where *func* is the name of the function you are calling. Write the preprocessor directive before any call to *func*.

Identifiers are mapped as described earlier in the chapter. Control is transferred as described earlier, by executing the instruction

```
BALR 14,15
```

with the address of the called function in R15. The called function must preserve the contents of R14 to return control to the C function that calls it. It may make use of R15 to address instructions and data within the called function. R13 points at an 18 word save area, on a fullword storage boundary, which the called function may use to store registers as described above.

If there are no arguments, R1 is set to zero by the calling function. Otherwise, R1 points to a special "argument block" which the calling function creates. To construct the argument block, the compiler processes arguments by the following rules:

* The lowest (first) entry in the argument block corresponds to the leftmost argument.
* The calling function converts each argument by the usual rules, as described in Chapter 5, "Expressions" in the section entitled "Function Calls," in your *C Language Manual* (SC09-1128).
* The compiler widens each argument to *int*, as described in Chapter 5, "Expressions" in the section entitled "Arithmetic Conversions," in your *C Language Manual* (SC09-1128). This conversion affects only arguments of an integer type represented in fewer than 32 bits.
* The calling function converts any argument of type *float* to type *double*.
* If the argument is a pointer type, the calling function adds it to the argument block unchanged.
* If the argument is not a pointer type, the calling function copies it into a temporary data object of the same type, and adds a pointer to the temporary data object to the argument block instead.
* The calling function sets the sign bit of the highest (last) entry in the argument block, which corresponds to the rightmost argument.

If you are calling a function that does not expect to be able to alter the value stored in any of its arguments, you can write

```
#pragma linkage(osfun, OS)

double osfun(double, double);
double x, y, z;

x = osfun(3.0, 2.7);
x = osfun(y, z);
```

Both function calls create temporary data objects and pass pointers to the temporary data objects in the argument block.

If you are calling an OS function that expects to be able to alter the value stored in any of its arguments and you call the function this way, the called function alters the value stored in the temporary data object, not the actual data object. To make sure that the value in the actual data object is altered, you must write:

```
#pragma linkage(osfunc, OS)

double osfunc(double *, double *);
double x, y, z;

x = osfunc(&y, &z);
```

The registers that need not be preserved by the called function are:

R0 – which is used to return values of integer type

R1 – which may be in any state

R14 – which may be in any state

R15 – which may be in any state

F0 – which is used to return values of type *double*

F2 – which may be in any state

F4 – which may be in any state

F6 – which may be in any state.

The called function again expects an 18 word save area, which R13 points at.

The called function returns a value of integer type in R0. All integer values must be converted to a 4 byte representation. The called function returns a value of floating type in F0. All floating values must be converted to an 8 byte representation, as for the C type *double*.

Most languages do not provide the ability to perform input/output and signal handling in more than one language. Generally, you perform I/O and signal handling from the language you start your program execution with.

Appendix A: Compiler Command Summary

(NO)ASM – Produce an object deck on **SYSLIN**. When you specify **NOASM**, the compiler writes assembler text to **SYSPUNCH** and does not generate an object deck. Default is **ASM**.

CSect(name) – Produce a control section (CSECT) with name *name*. When you specify **RENT** in combination with **CSECT** the compiler also produces a second CSECT with name **$***name*. *name* can be up to seven characters long when you specify **CSECT** in combination with **RENT**, and eight characters otherwise. The CSECT name you specify must not conflict with external function or data object names.

(NO)DEBug – Produce debugger entry points to enable linking with debugger modules. **DEBUG** is invalid in combination with **RENT**. **DEBUG** disables register allocation. Default is **NODEBUG**.

Define(name=def,name=def, ...) – Define a symbol at compile time of the form *name=def*. If you omit *=def*, the compiler assumes the definition to be "1". You may specify up to ten definitions.

(NO)HASM – **HASM** invokes Assembler-H. **NOHASM** invokes Assembler-XF. Default is to invoke the assembler specified as the default assembler when the compiler was installed.

(NO)IDENT8 – Restrict the number of significant internal identifiers to 8 characters. Default is **NOIDENT8** (31 characters).

(NO)LIBH – Strip the '**.h**' suffix from the name of **#include** files searched for as members of a partitioned dataset. When you specify **NOLIBH**, **#include** file names that end in the suffix '**.h**' correspond to partitioned dataset members with names ending in the suffix '**$H**'. Default is **NOLIBH**.

LINecount(n) – Use *n* as the number of lines to include in each page of a listing, including heading lines and blank lines. When you request a C source listing only, *n* can be any integer between 5 and 32767. When you request a listing that contains assembly language source, *n* can be any integer between 5 and 99 if you use Assembler-H or between 5 and 999 if you use Assembler-XF. Default is 60.

(NO)LISt and (NO)SOurce – If you specify **LIST**, the compiler produces an assembly language listing unless it detects errors. If the compiler detects errors when you specify **LIST** alone, it produces a C source listing interspersed with error messages. If you specify **SOURCE**, the compiler produces a C source and error listing. If you specify both **SOURCE** and **LIST**, the compiler produces an interspersed C source and assembly language listing unless it detects errors. If it detects errors when you specify both, the compiler produces a C source listing and an error listing. If you specify two or more options which conflict, and are therefore invalid, the compiler generates an error message and halts. The following options are invalid in the combinations listed:

PPONLY and **LIST**
PPONLY and **SOURCE**
LIST and **NOASM**

Default is **NOLIST** and **NOSOURCE**.

LSEarch(name) – Specify an alternate location for **#include** files you enclose in double quotation marks. *name* specifies one of the following:

1. The name of a partitioned dataset or
2. A DDname of the form **dd:***name*, where *name* is a DDname defining a partitioned dataset.

When you specify **LSEARCH**, the compiler first searches for the files you include in your program as members of the partitioned dataset or DDname you define with *name*. The compiler then searches for the files in the same partitioned dataset that it searches for **#include** files you enclose in angle brackets, as described under **SEARCH** below. Default is to search for **#include** files you enclose in double quotation marks in the same partitioned dataset the compiler searches for **#include** files you enclose in angle brackets.

(NO)MACarg – Expand macro arguments in string literals as the UNIX™ System V C compiler does. Default is **NOMACARG**.

(NO)PPonly – Execute the C preprocessor to expand all macros and show all include files in the C source, but do not compile the source. The compiler writes the expanded source to **SYSPUNCH** under MVS. Default is **NOPPONLY**.

(NO)REnt – Produce reentrant code by making every access to a data object through a base register. You must specify **CSECT** in combination with **RENT**. **DEBUG** is invalid in combination with **RENT**. Default is **NORENT**.

SEarch(name) – Specify an alternate location *name* for **#include** files enclosed in angle brackets **< >**. *name* specifies one of the following:

UNIX is a registered trademark of AT&T Bell Laboratories

1. The name of a partitioned dataset or

2. a DDname of the form **dd:***name*, where *name* is a DDname defining a partitioned dataset.

   When you specify **SEARCH**, the compiler first searches the partitioned dataset or DDname you specify with *name*. The compiler then searches the dataset defined by the DDname **INCLUDE**. **SEARCH** affects **#include** files enclosed in double quotation marks as discussed under **LSEARCH** above.

SEQ(m,n)|NOSEQ – Ignore line numbers from leftmost position *m* to rightmost position *n*. If you specify **NOSEQ** in combination with **SEQ(***m,n***)**, the compiler generates an error message and halts. Default is **NOSEQ** for variable length record format files and **SEQ(73,80)** for fixed record format files.

(NO)SHowinc – Show contents of include files in listing. Default is **NOSHOWINC**, which does not show the contents of **#include** files in listings.

(NO)TErminal – Use the **SYSTERM** dataset for compiler error output. When you specify **NOTERMINAL**, the compiler writes error output to **SYSPRINT** unless you request a listing. Default is **TERMINAL**.

(NO)UPCONV – Select UP (unsignedness preserving) rules for type conversion. Default is **NOUPCONV**, which selects value preserving rules as in the proposed ANSI standard.

Appendix B: Debugger Command Summary

Run program to next breakpoint	g
Run program to breakpoint *n*	*n*g
Step through program line by line	<ENTER>
Step through multiple (*n*) lines	*n* s
Exit debugging session	q
Print contents of a data object	:*data_object_name* p
Print contents using a pointer	:**pointer_name* p
Print character string using a pointer	:**pointer_name* p%s
Display current line	.
Display file name	f
Execute multiple debugger commands	*command*;*command*
Create log of debugger output	w *file_name*
Set breakpoint for a data object	*data_object_name* b [*cmd*]
Set a breakpoint for a line	*line_number* b
Delete breakpoint for a data object	*data_object_name* d
Delete a breakpoint for a line	*line_number* d
Change location by stack frames:	
up one frame	mu
down one frame	md
top of stack	mt
bottom of stack	mb
specific location	*location_specifier*m
Display list of stack frames from current frame to top of stack	t
List data object names and type for each frame	tv
List names, types, and addresses for each frame	tv&
Update value of specified location	u *new_value*
or	= *new_value*
List names and types for all data objects in current frame	v
List names, types, and addresses for all data objects in current frame	v&
Redirect command output to a file	>*file_name*
Redirect command input from a file	<*file_name*

Format Specifiers

One of the set of specifiers enclosed in braces is mandatory. Those enclosed in brackets are optional.

%[+|-| |#][.:>*nn*][.:>*nn*][h|l] {c|d|i|o|u|x|X|e|E|f|g|G|s|p}

A format specifier has the following parts:

* a percent sign **%**
* an optional flag +, –, #, or space
* an optional field width, as a decimal integer constant
* an optional decimal point followed by a precision, as a decimal integer constant
* an optional size qualifier **h**, **l**, or **L**
* a conversion character from the set

  {c d i o u x X e E f g G s p}

Type Casts

Type casts must take one of the forms

:(*data_type***)***data_element* p

or

:(*data_type***)***data_element* u

where *data_type* must be one of: {*char, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double*} and *data_element* must be a data object.

Appendix C: EBCDIC Character Table

Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC
00	NUL	10	DLE	20	DS	30	
01	SOH	11	DC1	21	SOS	31	
02	STX	12	DC2	22	FS	32	
03	ETX	13	DC3	23	WUS	33	IR
04	SEL	14	RES/ENP	24	BYP/INP	34	PP
05	HT	15	NL	25	LF	35	TRN
06	RNL	16	BS	26	ETB	36	NBS
07	DEL	17	POC	27	ESC	37	EOT
08	GE	18	CAN	28	SA	38	SBS
09	SPS	19	EM	29	SFE	39	IT
0A	RPT	1A	UBS	2A	SM/SW	3A	RFF
0B	VT	1B	CU1	2B	CSP	3B	CU3
0C	FF	1C	IFS	2C	MFA	3C	DC4
0D	CR	1D	IGS	2D	ENQ	3D	NAK
0E	SO	1E	IRS	2E	ACK	3E	
0F	SI	1F	IUS/ITB	2F	BEL	3F	SUB
40	SP	50	&	60	-	70	
41	RSP	51		61	/	71	
42		52		62		72	
43		53		63		73	
44		54		64		74	
45		55		65		75	
46		56		66		76	
47		57		67		77	
48		58		68		78	
49		59		69		79	‘
4A	¢	5A	!	6A	|	7A	:
4B	.	5B	$	6B	,	7B	#
4C	<	5C	±	6C	%	7C	@
4D	(	5D	)	6D	_	7D	’
4E	+	5E	;	6E	>	7E	=
4F		5F	¬	6F	?	7F	″

Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC
81	a	91	j	A1	~	B1	
82	b	92	k	A2	s	B2	
83	c	93	l	A3	t	B3	
84	d	94	m	A4	u	B4	
85	e	95	n	A5	v	B5	
86	f	96	o	A6	w	B6	
87	g	97	p	A7	x	B7	
88	h	98	q	A8	y	B8	
89	i	99	r	A9	z	B9	
8A		9A		AA		BA	
8B		9B		AB		BB	
8C		9C		AC		BC	
8D		9D		AD	[	BD	]
8E		9E		AE		BE	
8F		9F		AF		BF	
C0	{	D0	}	E0	\	F0	0
C1	A	D1	J	E1	NSP	F1	1
C2	B	D2	K	E2	S	F2	2
C3	C	D3	L	E3	T	F3	3
C4	D	D4	M	E4	U	F4	4
C5	E	D5	N	E5	V	F5	5
C6	F	D6	O	E6	W	F6	6
C7	G	D7	P	E7	X	F7	7
C8	H	D8	Q	E8	Y	F8	8
C9	I	D9	R	E9	Z	F9	9
CA	SHY	DA		EA		FA	]
CB		DB		EB		FB	
CC	graphic	DC		EC	chair	FC	
CD		DD		ED		FD	
CE	fork	DE		EE		FE	
CF		DF		EF		FF	EO

Index

#define 3-5
#include 3-5
#include files; location 3-4
#include files; search order 3-4
#pragma linkage directive 8-5
abnormal program termination 4-4
add library member 7-1
address of argument block 8-6
address of arguments 8-3
address specifiers; debugger 6-2
allocate a dataset 7-2
allocate files 3-5
allocate partitioned dataset 7-1
allocate space for library 7-2
allocation of calling environments 4-4
ANSI standard 4-8
ANSI; proposed standard 3-5
architecture; compiler 1-1
argc argument 5-1, 5-2
argument block address 8-6
argument data objects; pass 8-3
arguments of struct type; offset 8-3
arguments to macros 3-4
arguments; command 5-1, 5-2
arguments; start address 8-3
argv array 5-1, 5-2
arrays; declare 4-4
Assembler-H 3-3
Assembler-XF 3-3
assembly language listings 3-3
assembly language; call from C 8-1
assembly language; calls to 8-1
automatic register allocation 3-2
background execution 5-1
base address register 4-5
base register address 3-4
batch execution 5-2
batch mode; compiling in 4-2
batch processing 4-1
breakpoint; set and delete 6-6
C calling conventions 8-1
C programs; descriptive qualifier 2-2
C programs; file type 2-1
C programs; link under IMS 4-3
C runtime environment 4-6
C source listings 3-3
C stack 4-4
C#INIT1 control section 4-6
C; call from assembly language 8-1
call C from assembly language 8-1
calling conventions; C 8-1
calling environment; allocation 4-4
calling function; return control to 8-5
calls to assembly language 8-1
character set 2-1
columns; ignore 3-5
command arguments 5-1, 5-2
command summary; debugger 6-2
compile programs for IMS 4-3
compiler architecture 1-1
compiler options 3-2
compiler; introduction 1-1
compiling in batch mode 4-2
compiling under ISPF 3-7
compiling under TSO 3-5
control section 3-2, 4-5, 4-6
control; return to caller 8-5
control; transfer 8-2
create library 7-1
CSECT 3-2, 4-5
CSECT option 4-5
data object; static 4-5
data objects; examine 6-2
data objects; global 6-2
data objects; pass 8-3
dataset; allocate 7-2
DD names 3-1
DD statements 3-6
debugger command output; log of 6-6
debugger entry points 3-2
debugger; address specifiers 6-10
debugger; breakpoints 6-6
debugger; command summary 6-2
debugger; display line and file 6-5
debugger; enable 6-1
debugger; examine data objects 6-4
debugger; get function name and line 6-2
debugger; get source line contents 6-2
debugger; multiple commands 6-5
debugger; single step execution 6-3
debugging 3-2
debugging; type casts 6-9
declare arrays 4-4

declare functions with external linkage 8-1
define identifier 3-2
delete breakpoint; debugger 6-2
delete library member 7-2
descriptive qualifier for C programs 2-2
descriptive qualifiers 4-1
dummy sections; external 4-5
echo.c source listing 6-1
edit source files 2-1
enter programs in ISPF 2-2
enter programs in TSO 2-2
entry point 7-1
entry point; under IMS 4-3
entry points; debugger 3-2
error messages 3-3
examine data objects 6-2, 6-4
example source code 6-1
EXEC statements 3-6
execute load modules 5-1
execute object modules 5-1
execution in background 5-1
execution under ISPF 5-1
execution under TSO 5-1
execution; batch 5-2
expansion of macros 3-4
external dummy sections 4-5
external identifiers 3-3
external linkage; declare functions 8-1
external names; form 8-1
external names; map from C identifiers 8-1
FBS record format 4-6
file allocation 3-5
file inclusion 3-4
file type for C programs 2-1
files; fixed record format 3-5
files; variable length record format 3-5
fixed record format 4-6
fixed record format files 3-5
floating values 8-3
fopen function 4-6
foreground processing 4-1
form external names 8-1
function; return control to caller 8-5
get function name; debugger 6-2
get source line; debugger 6-2
global data objects 6-4
holes 8-3
identifier names; map 8-6
identifiers; external 3-3
identifiers; internal 3-3
identifiers; map to external names 8-1
IEBGENER program 4-6
IMS; compile programs for 4-3
IMS; entry point 4-3
IMS; link C programs under 4-3
INCLUDE 3-1
include files 3-4
include object modules 7-2
increase stack size 4-4
indirection; debugger 6-2
integer values 8-3
internal identifiers 3-3
introduction; compiler 1-1
ISPF; compiling under 3-7
ISPF; enter programs 2-2
ISPF; execute programs under 5-1
ISPF; link under 4-1
JOB statements 3-6
length of line 2-1
levels of indirection 6-9
library member; delete 7-2
library member; replace 7-3
library members; delete 7-2
library members; link 7-2
library members; list 7-3
library; add member 7-1
library; allocate space for 7-2
library; create 7-1
library; object module 4-1
library; runtime 4-1
line length 2-1
link C programs under IMS 4-3
link edit object module 7-1
link library members 7-2
Link Pack area 4-5
Link Pack Area 4-7
link under ISPF 4-1
link under TSO 4-1
linkage editor 1-1
linkage editor options 4-2
list library members 7-3
listing options 3-3
listings 1-2, 3-5
listings; assembly language 3-3
listings; C source 3-3
load module 4-1
load module; execution 5-1
load programs 4-1
load programs under TSO 4-1
log command output; debugger 6-2
logical record length; source files

2-1
lrecl; values 2-1
macro arguments 3-4
macro expansion 3-4
map C identifiers to external names 8-1
mapping identifier names 8-6
multiple commands; debugger 6-2
NAME statement; terminate 7-1
no format error diagnostic 6-10
object module 3-1, 3-2
object module library 4-1
object module; execution 5-1
object module; link edit 7-1
object modules; include 7-2
options to the compiler 3-2
OS calling sequence 8-1, 8-5
partitioned dataset; allocate 7-1
pass argument data objects 8-3
pointer values 8-3
preprocessor 3-4
preprocessor directive; #pragma linkage 8-5
preserve registers 8-2
processing; background 4-1
processing; foreground 4-1
program entry; instructions 8-3
program termination; abnormal 4-4
programs in C; file type 2-1
programs; load 4-1
proposed ANSI standard 3-5
recfm; values 2-1
record format; source files 2-1
reentrancy 3-2
reentrant code 3-4
reentrant load module 4-7
reentrant program 4-4
register allocation; automatic 3-2
registers; preserve 8-2
RENT option 4-5
replace library member 7-3
restrictions on source files 2-1
return a value 8-5
return control to caller 8-5
RLINK program 4-5, 4-6
rules; unsignedness preserving 3-5
rules; value preserving 3-5
runtime environment 4-6
runtime library 4-1
save area 8-2
search modifier 3-4
search order 3-5, 4-2
set breakpoint; debugger 6-2
signal SIGSTACK 4-4
significant characters 3-3
SIGSTACK signal 4-4
size of stack 4-4
source code; example program 6-1
source files; edit 2-1
source files; logical record length 2-1
source files; record format 2-1
source files; restrictions 2-1
source input 3-1
stack 4-4
stack area 8-4
stack overflow 4-4
stack size 4-4
stack size; increase 4-4
standard error stream 5-1
standard input stream 5-1
standard output stream 5-1
standard text streams 5-2
static data object 4-5
stderr text stream 5-1
stdin text stream 5-1
stdout text stream 5-1
struct arguments; offset 8-3
SUBMIT command 3-7
SYSERR 5-2
SYSIN 3-1, 5-2
SYSLIN 3-1
SYSPRINT 3-1, 5-2
SYSPUNCH 3-1
SYSTERM 3-1
SYSUT1 3-1
SYSUT2 3-1
SYSUT3 3-1
terminate NAME statement 7-1
termination; abnormal 4-4
text streams; standard 5-2
transfer control to function 8-2
trigraphs 2-1
TSO; compiling under 3-5
TSO; enter programs 2-2
TSO; execution under 5-1
TSO; link under 4-1
TSO; loading programs under 4-1
type casts in debugging 6-9
type casts; debugging 6-2
unsignedness preserving rules 3-5
value preserving rules 3-5
values; lrecl 2-1

values: recfm 2-1
values; return 8-5
variable length record format files
3-5

International Business Machines Corporation

Order Number SC09-1129-01

File Number S370-99

Printed in U.S.A.